Case Study: Pthread Synchronization

Dr. Yingwu Zhu

Thread Mechanisms

- Birrell identifies four mechanisms commonly used in threading systems
  - Thread creation
  - **Mutual exclusion (mutex)**
  - **Waiting for events - condition variables**
  - Interrupting a thread's wait
- First three commonly used in thread systems
- Take home message: Threads programming is tricky stuff! Stick to established design patterns.

Thread Creation in PThreads

- Type: pthread_t tid;   /* thread handle */
- pthread_create (&tid, thread_attr, start, arg)
  - tid returns pointer to created thread
  - thread_attr specifies attributes, e.g., stack size; use NULL for default attributes
  - start is procedure called to start execution of thread
  - arg is sole argument to proc
  - pthread_create returns 0 if thread created successfully
- pthread_join (tid, &retval);
  - Wait for thread tid to complete
  - Retval is valued returned by thread
- pthread_exit(retval)
  - Complete execution of thread, returning retval

Example

```
#include<pthread.h>
#include <stdio.h>
/* Example program creating thread to compute square of value */
int value;/* thread stores result here */
void* my_thread(void *param);        /* the thread */
main (int argc, char *argv[])
{     pthread_t tid;                 /* thread identifier */
      int retcode;/* check input parameters */
      if (argc != 2) {   fprintf(stderr,"usage: a.out <integer value>\n"); exit(1);  }
      /* create the thread */
      retcode = pthread_create(&tid,NULL,my_thread,argv[1]);
      if (retcode != 0)   {     fprintf(stderr,"Unable to create thread\n"); exit (1); }
      /* wait for created thread to exit */
      pthread_join(tid,NULL);
      printf ("I am the parent: Square = %d¥n", value);}
      /* The thread will begin control in this function */

void *my_thread(void *param)
{
      int i = atoi (param);
      printf ("I am the child, passed value %d\n", i);
      value = i * i;
      /* next line is not really necessary */
      pthread_exit(0);
}
```

Mutual Exclusion

- Bad things can happen when two threads "simultaneously" access shared data structures: Race condition → critical section problem
  - Data inconsistency!
  - These types of bugs are really nasty!
    - Program may not blow up, just produces wrong results
    - Bugs are not repeatable
- Associate a separate lock (mutex) variable with the shared data structure to ensure "one at a time access"

Mutual Exclusion in PThreads

- pthread_mutex_t   mutex_var;
  - Declares mutex_var as a lock (mutex) variable
  - Holds one of two values: “locked” or “unlocked”
- pthread_mutex_lock (&mutex_var)
  - Waits/blocked until mutex_var in unlocked state
  - Sets mutex_var into locked state
- pthread_mutex_unlock (&mutex_var)
  - Sets mutex_var into unlocked state
  - If one or more threads are waiting on lock, will allow one thread to acquire lock

Example:

```
                                  //pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t   m;   //pthread_mutex_init(&m, NULL);
...
pthread_mutex_lock (&m);
<access shared variables>
pthread_mutex_unlock(&m);
```

Problems with Mutex Locks

- A scenario:
  - Producer threads enqueue new items
  - Consumer threads dequeue items
  - Race condition: the queue
- What is the potential problem?

Pthread Semaphores

- #include <semaphore.h>
- Each semaphore has a counter value, which is a non-negative integer

Pthread Semaphores

- Two basic operations:
  - A *wait operation decrements the value of the semaphore by 1. If the value is* already zero, the operation blocks until the value of the semaphore becomes positive (due to the action of some other thread).When the semaphore's value becomes positive, it is decremented by 1 and the wait operation returns. → sem_wait()
  - A *post operation increments the value of the semaphore by 1. If the semaphore* was previously zero and other threads are blocked in a wait operation on that semaphore, one of those threads is unblocked and its wait operation completes (which brings the semaphore's value back to zero). → sem_post()

Slightly different from our discussion on semaphores

Pthread Semaphores

- sem_t   s; //define a variable
- sem_init(); //must initialize
  - 1st para: pointer to sem_t variable
  - 2nd para: must be zero
    - A nonzero value would indicate a semaphore that can be shared across processes, which is not supported by GNU/Linux for this type of semaphore.
  - 3rd para: initial value
- sem_destroy(): destroy a semaphore if do not use it anymore

Pthread Semaphores

- int sem_wait(): wait operation
- int sem_post(): signal operation
- int sem_trywait():
  - A nonblocking wait function
  - if the wait would have blocked because the semaphore's value was zero, the function returns immediately, with error value EAGAIN, instead of blocking.

Example

```c
#include <malloc.h>
#include <pthread.h>
#include <semaphore.h>
struct job {
/* Link field for linked list. */
struct job* next;
/* Other fields describing work to be
    done...*/
};
/* A linked list of pending jobs. */
struct job* job_queue;
/* A mutex protecting job_queue. */
pthread_mutex_t job_queue_mutex =
    PTHREAD_MUTEX_INITIALIZER;
```

```c
/* A semaphore counting the number of jobs in the queue. */
sem_t job_queue_count;
/* Perform one-time initialization of the job queue. */
void initialize_job_queue ()
{
        /* The queue is initially empty. */
        job_queue = NULL;
        /* Initialize the semaphore which counts jobs in the
        queue. Its initial value should be zero. */
        sem_init (&job_queue_count, 0, 0);
}
```

Assume infinite queue capacity.

Example

```
/* Process dequeued jobs until the queue is empty. */
void* thread_function (void* arg)
{
while (1) {
        struct job* next_job;
    /* Wait on the job queue semaphore. If its value is positive,indicating that the queue is not empty,
       decrement the count by 1. If the queue is empty, block until a new job is enqueued. */
     sem_wait (&job_queue_count);
    /* Lock the mutex on the job queue. */
     pthread_mutex_lock (&job_queue_mutex);
    /* Because of the semaphore, we know the queue is not empty. Get the next available job. */
     next_job = job_queue;
    /* Remove this job from the list. */
     job_queue = job_queue->next;
    /* Unlock the mutex on the job queue because we’re done with the queue for now. */
     pthread_mutex_unlock (&job_queue_mutex);
    /* Carry out the work. */
     process_job (next_job);
    /* Clean up. */
     free (next_job);
 }
return NULL;
}
```

Example

```
/* Add a new job to the front of the job queue. */
void enqueue_job (/* Pass job-specific data here... */)
{
    struct job* new_job;
    /* Allocate a new job object. */
    new_job = (struct job*) malloc (sizeof (struct job));
    /* Set the other fields of the job struct here... */
    /* Lock the mutex on the job queue before accessing it. */
    pthread_mutex_lock (&job_queue_mutex);
    /* Place the new job at the head of the queue. */
    new_job->next = job_queue;
    job_queue = new_job;
    /* Post to the semaphore to indicate that another job is available. If
    threads are blocked, waiting on the semaphore, one will become
    unblocked so it can process the job. */
    sem_post (&job_queue_count);
    /* Unlock the job queue mutex. */
    pthread_mutex_unlock (&job_queue_mutex);
}
```

Can they switch order?

Condition Variables

Waiting for Events: Condition Variables

- Mutex variables are used to control access to shared data
- Condition variables are used to wait for specific events
  - Buffer has data to consume
  - New data arrived on I/O port
  - 10,000 clock ticks have elapsed

Let's see an example

```
void* thread_function (void* thread_arg)
{
  while (1) {
    int flag_is_set;

    /* Protect the flag with a mutex lock.  */
    pthread_mutex_lock (&thread_flag_mutex);
    flag_is_set = thread_flag;
    pthread_mutex_unlock (&thread_flag_mutex);

    if (flag_is_set)
      do_work ();
    /* Else don't do anything.  Just loop again.  */
  }
  return NULL;
}

/* Sets the value of the thread flag to FLAG_VALUE.  */

void set_thread_flag (int flag_value)
{
  /* Protect the flag with a mutex lock.  */
  pthread_mutex_lock (&thread_flag_mutex);
  thread_flag = flag_value;
  pthread_mutex_unlock (&thread_flag_mutex);
}
```

The thread execution is controlled by a flag

If we use mutex locks only, what happens?

Condition Variables

- Enable you to implement a condition under which a thread executes and, inversely, the condition under which the thread is blocked

Condition Variables in PThreads

- pthread_cond_t   c_var;
  - Declares **c_var** as a condition variable
  - Always associated with a **mutex** variable (say m_var)
- pthread_cond_wait (&c_var, &m_var)
  - Atomically unlock **m_var** and block on **c_var**
  - Upon return, mutex m_var will be re-acquired
  - Spurious wakeups may occur (i.e., may wake up for no good reason - always recheck the condition you are waiting on!)
- pthread_cond_signal (&c_var)
  - If no thread blocked on **c_var**, do nothing
  - Else, unblock a thread blocked on c_var to allow one thread to be released from a pthread_cond_wait call
- pthread_cond_broadcast (&c_var)
  - Unblock all threads blocked on condition variable c_var
  - Order that threads execute unspecified; each reacquires mutex when it resumes

Waiting on a Condition

```
pthread_mutex_t
    m_var=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c_var=PTHREAD_COND_INITIALIZER;
//pthread_cond_init()
pthread_mutex_lock (m_var);
while (<some blocking condition is true>)
      pthread_cond_wait (c_var, m_var);
<access shared data structrure>
pthread_mutex_unlock(m_var);
```

Note: Use “while” not “if”; Why?

Revisit on the example

```
void* thread_function (void* thread_arg)
{
  /* Loop infinitely.  */
  while (1) {
    /* Lock the mutex before accessing the flag value.  */
    pthread_mutex_lock (&thread_flag_mutex);
    while (!thread_flag)
      /* The flag is clear.  Wait for a signal on the condition
         variable, indicating that the flag value has changed.  When the
         signal arrives and this thread unblocks, loop and check the
         flag again.  */
      pthread_cond_wait (&thread_flag_cv, &thread_flag_mutex);
    /* When we've gotten here, we know the flag must be set.  Unlock
       the mutex.  */
    pthread_mutex_unlock (&thread_flag_mutex);
    /* Do some work.  */
    do_work ();
  }
  return NULL;
}

/* Sets the value of the thread flag to FLAG_VALUE.  */

void set_thread_flag (int flag_value)
{
  /* Lock the mutex before accessing the flag value.  */
  pthread_mutex_lock (&thread_flag_mutex);
  /* Set the flag value, and then signal in case thread_function is
     blocked, waiting for the flag to become set.  However,
     thread_function can't actually check the flag until the mutex is
     unlocked.  */
```

Example continued…

```
  thread_flag = flag_value;
  pthread_cond_signal (&thread_flag_cv);
  /* Unlock the mutex.  */
  pthread_mutex_unlock (&thread_flag_mutex);
}
```

Exercise

- Design a multithreaded program which handles bounded buffer problem using semaphores
  - int buffer[10]; //10 buffers
  - rand() to produce an item
  - int in, out;
  - Implement producers and consumers process